Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Abhängige Metriken für on-the-fly Analyse von kritischen Pfaden

Dependent Metrics for On-the-Fly Critical Path Analysis

Bachelorarbeit

Leon Streichardt
Matrikelnummer: 434838

Aachen, den 10. April 2026

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (’)
Zweitgutachter:	Dr. rer. nat. Stefan Lankes (*)
Betreuer:	Ben Thärigen, M.Sc. (’)

(’) Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Lehrstuhl für Betriebssysteme, RWTH Aachen University

communicated by Prof. Matthias S. Müller

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 10. April 2026

Kurzfassung

Das On-The-Fly-Critical-Path-Tool (OTF-CPT) ist bereits in der Lage, den kritischen Pfad eines Programms während der Ausführung zu berechnen. Jedoch ist diese Analyse auf die Laufzeit des kritischen Pfads limitiert, was in den meisten Fällen jedoch nicht genug Kontext bietet, um die beobachtete Leistung nachzuvollziehen.

In dieser Arbeit wird das OTF-CPT durch eine neue Schnittstelle erweitert, welche es dem Nutzer erlaubt beliebige Metriken entlang des kritischen Pfads zu messen. Diese Schnittstelle wird unter benutzung von PAPI implementiert, um beliebige Hardware Counter messen zu können.

Daraufhin folgt eine Evaluation, welche sich genauer mit dem Overhead der durch diese Erweiterung entstanden ist auseinandersetzt und vergleicht zudem die gemessenen Werte für die Hardware Counter mit Ergebnissen aus anderen tools wie Score-P und Scalasca. Die Evaluation zeigt, dass obwohl der Overhead sehr verschieden ausfallen kann, je nachdem, was für ein Programm gemessen wird und wie viele Metriken gemessen werden, ist er meistens trotzdem in einem akzeptablen Bereich. Ebenfalls zeigen die Ergebnisse, dass die Werte, die man mit dieser neuen Version des OTF-CPT, erhält vergleichbar sind mit den Werten die man durch anderer Tools erhält und diese genutzt werden können, um die Leistungs-Eigenschaften eines Programms zu characterisieren.

Stichwörter HPC, OpenMP, MPI, PAPI, Critical-Path

Abstract

The On-The-Fly-Critical-Path-Tool (OTF-CPT) can already perform the critical path analysis online during the execution of the measured application. However, the analysis is limited to only track the execution time of the critical path, which often is not enough context to understand the observed performance of an application.

This thesis extends the OTF-CPT by introducing a new interface to allow for tracking arbitrary metrics on the critical path. The new interface is then implemented to support the tracking of hardware counters using PAPI.

An evaluation is done that investigates the overhead of the added capabilities and compares the results from OTF-CPT with results from other tools, like Score-P and Scalasca. The evaluation shows that although the overhead can vary depending on the characteristics of the measured application and how many metrics are being tracked, it is acceptable in most cases. It also shows that the results obtained using the new metrics are comparable to the results from other tools and how they could be used to characterise the performance of an application.

Keywords: HPC, OpenMP, MPI, PAPI, Critical-Path

Contents

List of Figures xi

1 Introduction 1

2 Background 3
- 2.1 OpenMP . . . 3
- 2.2 MPI . . . 4
- 2.3 Critical Paths . . . 5
- 2.4 PAPI . . . 6

3 Concept and Implementation 9
- 3.1 OTF-CPT . . . 9
- 3.2 Dependent Metrics . . . 10
  - 3.2.1 Storage of Dependent Metrics . . . 11
  - 3.2.2 Functions of the interface . . . 11
  - 3.2.3 Interface implementation for PAPI . . . 14
- 3.3 Limitations . . . 14

4 Evaluation and Results 17
- 4.1 Setup . . . 17
- 4.2 Overhead . . . 17
- 4.3 Comparison to existing results . . . 21

5 Conclusion and Future Work 25

Bibliography 27

List of Figures

2.1 Critical Path example 5

3.1 Interface for metrics being tracked 12

4.1 Overhead depending on the number of threads used (jukkr-miniapp openmp) 18
4.2 Overhead depending on the number of threads used (poc-jukkr openmp) 18
4.3 Overhead for ScoreP Run (poc-jukkr) 19
4.4 Overhead depending on the number of supported metrics (no metrics used, 8 threads, poc-jukkr) 20
4.5 Overhead depending on the number of used metrics (but same number of supported metrics, 8 threads, poc-jukkr) 21
4.6 Score-P measurements (poc-jukkr) 22
4.7 OTF-CPT measurements for entire application (poc-jukkr) 22
4.8 OTF-CPT measurements along the critical path (poc-jukkr) 23

1 Introduction

High performance computing applications are ever-growing and increasing in complexity. At the same time the need to achieve the highest possible performance on the underlying hardware is increasing as the demand for compute is rising. This creates a complicated situation for optimizing the performance of an application as it can be difficult to fully understand all the interactions and what exactly may be the bottleneck.

Traditional tools typically perform the critical path analysis as part of a post-processing step. The most prominent tooling for this purpose would be a combination of Score-P[8] and Scalasca[12]. In this case, Score-P is used to collect a trace during the execution, which contains all the raw information needed to reconstruct the execution in later analysis steps. Scalasca is then used to perform the post-processing of the traces collected by Score-P and extract the information of interest, like the critical path and related information. These tools typically focused mostly on just the execution time of the critical path, however, this provides no actionable information about what might be the cause for the observed execution time. This limitation is already being addressed in tools like Scalasca, by trying to find better performance indicators during their analysis [1].

The work in this thesis builds on the OTF-CPT[11], which can already perform the critical path analysis online. This analysis is currently limited to only track the execution time of the critical path being tracked. However, only knowing the execution time is often not enough context to properly understand the performance behaviour of the application at hand and is especially limiting for trying to improve the performance, as there is no indication of what the work should focus on.

This thesis will attempt to extend the critical path analysis of the OTF-CPT, by adding the ability to track arbitrary metrics along the critical path. Specifically the OTF-CPT performs the critical path analysis online, as before, and is collecting arbitrary metrics on the critical path as well. The available metrics to track as part of this thesis will be limited to hardware counters using PAPI, but there is potential to extend the tool with more metrics in the future.

The goal with the new metrics would be to provide more information and context about the execution of the critical path to help with understanding and improving performance.

Chapter 2 will cover the necessary background information to follow the rest of the thesis. Then the requirements and goals regarding the dependent metrics will be discussed in Chapter 3.2 and a new interface in the OTF-CPT to track arbitrary metrics is proposed in Chapter 3.2.2. Afterwards in Chapter 3.2.3 an implementa-

tion of the proposed interface to track PAPI metrics is showcased and some of the limitations of the new approach are discussed in Chapter 3.3. This will be followed by an evaluation in Chapter 4 focusing on the overhead introduced by the OTF-CPT and the addition of the dependent metrics as well as comparing the measured values to other tools. The thesis then wraps up with an overall conclusion and an outlook for potential future work and improvements to be made in Chapter 5.

2 Background

This section will introduce required background knowledge for this thesis, including OpenMP, MPI, PAPI and the concept of a critical path.

2.1 OpenMP

OpenMP[10, 3] is a software API specification, which aims to define a portable, scalable model with a simple and flexible interface for developing parallel applications. OpenMP can be used on a variety of platforms ranging from the desktop to the supercomputer and is used for shared-memory parallel programming in C/C++ and Fortran. Most compilers provide an implementation of this specification to easily use OpenMP.

The design principle behind OpenMP is to make use of worksharing constructs, to split up work into smaller, mostly self-contained parts and then efficiently executing these parts. The two main approaches to split up an application using OpenMP are data parallelism and task parallelism. One example for data parallelism would be the *parallel-for*, which splits up a normal for-loop into multiple computations that each are responsible for a specific range of the original for-loop. These smaller computations can then be spread across different threads for increased parallelism. The alternative would be to use task parallelism by manually breaking up the work into separate tasks. A task in OpenMP represents a single unit of work and may be executed immediately, deferred or run on another thread to use more of the available hardware. Regardless of how one uses OpenMP, it will always be done by using specific pragmas to annotate the application's code. This will then automatically be converted into the corresponding implementation details by the toolchain.

Importantly for this work, OpenMP provides an API, called OMPT[4], which allows tools to register callbacks for a variety of events that occur during execution. This allows tools to execute their own code to handle certain events, like when a task is being created or started, as well as things like synchronization points being reached. This is important for this work, as synchronization points, like barriers, represent the moment in which different threads reach a common point in their execution. In most cases, this also means that one logical block of useful computation in the application is about to start or was just completed, depending on the kind of synchronization.

2.2 MPI

MPI[7] (Message-Passing Interface) is a message-passing API specification, which is a widely used standard for writing message-passing programs.

MPI allows different instances of an application to communicate with each other using messages. These processes could be running on the same machine or on a different machine, which is why this work will refer to each instance of an application as a *node*.

As already mentioned, MPI allows multiple nodes to communicate with each other via messages. A message conforms to some pre-determined format and acts as the smallest unit from the user's point of view. This means at least from a user perspective one can always deal with entire messages without having to consider the underlying implementation details, which simplifies the development of applications and provides more flexibility as the underlying transport layer can be changed without modifying the applications.

MPI supports three main kinds of communication: point-to-point, one-sided and collectives.

Point-to-point represents the simplest type of communication, which is just direct communication between two nodes. Whenever these two nodes are communicating with each other via send and receive, they introduce a dependency and a kind of synchronization between the sender and receiver.

Collectives are a kind of communication, where all nodes of a communicator are taking part in. Typically, a collective is used to exchange information between all nodes and compute some result based on all the data. One such example would be a reduction with a sum operation, which will involve all nodes contributing their data and computing the sum as a whole. Just like with point-to-point communication, using such a collective introduces a dependency and kind of synchronization between all nodes of a communicator, as they need to coordinate their exchange.

One-sided communication allows for transferring data without directly involving the other node in every transfer. Compared to point-to-point communication, where there is always a send-recv pair. One-sided communication allows a node to directly read specific data from another node's memory and to directly put some data into the other node's memory as well. This does not provide any kind of dependency or synchronization itself, but instead it is usually recommended to synchronize manually before and after intending to exchange data.

In contrast to OpenMP, MPI allows tools to directly intercept the calls made by the application, by overwriting the default functions prefixed with *MPI_*. This means that the tool has full control to do whatever it needs to and will then likely forward the call or a modified version of the call to the underlying MPI implementation, by calling one of the functions prefixed with *PMPI_*. For example, if a tool wants to gather information about the number of messages sent by an application. It would overwrite the *MPI_Send* function with their own version that would increment the counter for the number of messages sent and then forwards the call to the underlying implementation by calling *PMPI_Send*.

2.3 Critical Paths

A critical path[14] is a property that emerges in any given execution of a parallel program and can vary depending on the input to the application as well as the general context in which the application is executed.

Parallel programs are often constructed by breaking a single serial computation into multiple smaller serial computations that can work on different parts of the problem domain at a time. These smaller computations can then be executed in parallel across multiple threads or processes if they are independent of each other. However, in most cases there are dependencies between the different computations as the output of one computation is likely to be the input to a different computation, which requires synchronization between them and introduces a dependency between different steps. In practice these dependencies could manifest as send and receive pairs or collective operations for MPI. And when using OpenMP these dependencies would be introduced by barriers, which could be either explicit barriers or implicit barriers like the ones at the end of a parallel-for.

We can consider a path in a specific execution as a chain of computations, where each computation needs to wait for the previous one in the chain to complete before it can start executing. The critical path is then defined as the path with the longest overall computation time. However, what exactly this computation time includes can vary, the OTF-CPT supports tracking the so called useful-computation time, which is only the time spent in the application code itself, as well as the useful-computation time + the time spent in OpenMP or MPI. The critical path can then also be further constrained to only be computed per process or per thread, instead of covering the entire application.

The application level critical path determines the execution time of the entire application, as every part of it needs to finish and by definition has the longest total execution time. This also means that, if one wants to improve the overall runtime of the application one needs to focus on the parts that are on this critical path, as this determines the runtime and improving the performance of other aspects is unlikely to yield any benefit.

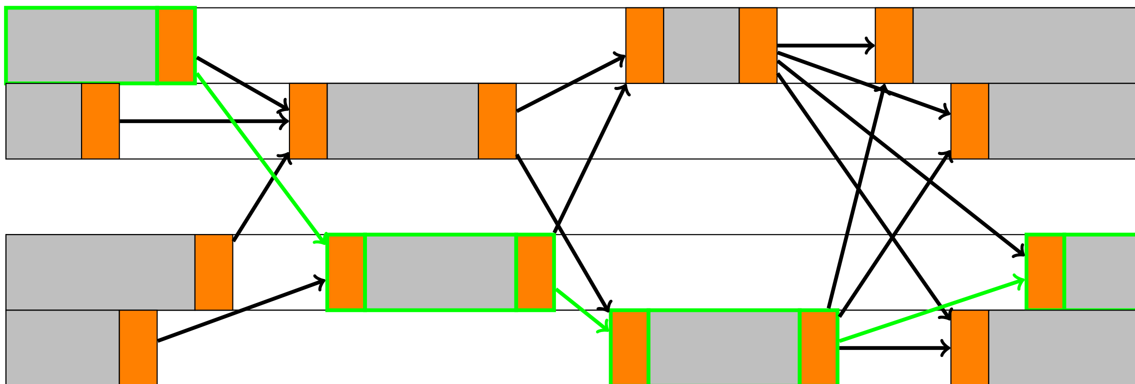

Figure 2.1: Critical Path example

One common way that tools use to determine the critical path, is to only collect traces during the execution and then after the application has terminated, the collected data can be used in a post-processing step to calculate the critical path. The traces collected during execution allow tools to build up a timeline of the execution and then perform the analysis backwards while having essentially perfect knowledge of the execution. One example for such a timeline can be seen in figure 2.1 in which there are two processes with two threads of execution each. The light gray sections indicate the useful-computation time whereas the orange parts indicate time spent in OpenMP or MPI and, the arrows between the different blocks indicate some form of synchronization, inducing a dependency between them. The computed critical path is then marked in green.

To obtain the application's critical path from the example, one would see that thread 1 of process 2 is the last to reach the end, so it will be the starting point of our analysis. From there, one can look for the computation that it was waiting on, which in this case would be the one from thread 2 of process 2. This procedure is then repeated until the beginning of the application is reached. Obtaining process level critical paths would then be the same general procedure, with the only difference being that one only looks at one process at a time and ignores the others.

2.4 PAPI

PAPI[9] (Performance Application Programming Interface) is an interface to access hardware specific counters. Hardware counters are special registers in the hardware, which allow for tracking the number of times a certain action has been performed or event has occurred. This, alongside other tools, can help give a better understanding of what exactly is happening on the hardware level to explain the observed performance of an application and guide developers towards potential improvements. Commonly used counters include the number of instructions executed and the number of memory and cache accesses, hits and misses. However, as they rely on specific hardware, the number of these counters that can be active at any given time is limited to a small number, usually around 8 for server CPUs [5].

Furthermore, in most cases where hardware counters are being used for working on an application, one does not directly use PAPI itself in the application, but instead makes use of other tools, like Score-P[8], which will then use PAPI internally to interact with these counters. If one does use PAPI directly as a library, it provides two main interfaces for obtaining values from hardware counters. One is a high-level interface [13], which allows the developers to quickly measure hardware counters during certain regions but provides little control outside of that. This is done by having the developer insert calls that mark the start and end of regions, in which hardware counter should be tracked. However, this work mainly makes use of the second, low-level interface. This is more work to integrate, as one needs to configure all the counters manually and is responsible for starting, stopping and reading the

counters, but this also provides more control. The main issue with the high-level API for this use-case, stems from the need to explicitly mark the start and end of regions which should be measured. However, the tool needs to start the counters at the beginning of the execution, get the values at specific moments during it and then stop them only at the end. This makes the high-level API not suitable for use in the OTF-CPT.

3 Concept and Implementation

The next section will cover the general ideas, concepts and motivation that guided the implementation and will address some of the details regarding the final implementation used. The end goal is to be able to track dependent metrics on the critical path. Dependent metrics in this case are just arbitrary metrics that specifically are tracked along the critical path, but they do not influence how the critical path is calculated.

3.1 OTF-CPT

OTF-CPT (On-the-fly Critical Path Tool) is a tool that allows basic tracking of the critical path of an application during runtime. It was first developed as part of this publication [11] by Joachim Protze and was then further developed in by Joachim Jenke [6].

Compared to other tools that determine the critical path as part of a post-processing step, the OTF-CPT determines the length of the critical path during the execution of the application and can therefore provide the results immediately after the application itself terminates. This has the benefits of providing faster results, as there are no extra steps involved to get the desired information, and also avoids storing large traces that would be needed for the post-processing steps. However, this approach also means that the information available to the tool is more limited compared to a post-processing step, as it does not know the rest of the execution yet. Previously this meant that the tool could only provide information about the execution time of the different critical paths of the application, like useful-computation based, but nothing else. After this work, it can also provide some basic performance related metrics, in the form of hardware counters as well. This initial support already allows the users of the tool to obtain information like IPC or cache hits and misses along the critical path. However, the results are presented only as the final numbers with no detailed breakdown available, which can seem very limited compared to other performance related tooling. These limitations will be discussed in more detail in the following limitations section 3.3.

In order to track the critical path, the OTF-CPT needs to use a different approach compared to the more widespread approach that was also covered in the background section of this work. The main issue comes from the fact, that the usual approaches work backwards through the execution of the application, however the OTF-CPT needs to determine the critical path going forward, while the application is running and with no way of looking back.

The main idea behind the approach used by the OTF-CPT is to locally keep track of the best currently known idea of the critical path. This means that every thread in every process knows what the critical path would be if the application were to terminate at that point in time. This is done by keeping track of the currently known execution time of the critical-path on a per-thread basis. Whenever threads or processes then synchronize or communicate using OpenMP or MPI, this information is exchanged and the threads and processes update their local knowledge by choosing the larger execution time and using this as their new starting point when continuing the computation. Even though every thread keeps their own version, we are guaranteed to reach a global consensus at some point, assuming normal execution. As every thread and process will synchronize at the end of the execution in OpenMPs or MPIs shutdown procedures, this will give a last chance to perform the exchange of critical path information.

To implement this approach, the tool stores the state in thread-local instances of the "ThreadClock" class, which contains multiple different clocks to keep track of the thread, process and application specific critical paths along with different versions for useful-computation, useful-computation including MPI and useful-computation including OpenMP. The tool then registers callbacks for OpenMP to get called for any important actions perform by OpenMP, like starting new threads, entering a parallel block, exiting a parallel block and many more of such synchronization points. Whenever one of these callbacks is called, the tool will then either start keeping track of the time or stop the current time keeping and update the thread-local state accordingly. For MPI this is fundamentally the same approach, but instead of registering callbacks, the tool overwrites the default MPI_ functions with versions that perform the necessary bookkeeping and then forward the call to the underlying implementation using the PMPI_ version of the called function.

3.2 Dependent Metrics

Although this thesis mainly focuses on tracking hardware counters as dependent metrics, the design of the implementation should be able to accommodate other metrics as well for future usage. Because of this, a new interface was introduced which abstracts away the specific metrics being tracked and provides a simple interface for the rest of the tool to use. This should allow for a variety of metrics to be supported in the future, with little to no changes being necessary outside the implementation of the interface for said metrics as well as some of the set up related code, which would need to know how to correctly configure the new metrics.

The general requirements for the interface are to provide the start and end values of the metrics and some bookkeeping related functions such as getting the names of the metrics to properly display results at the end and to notify the metrics implementation about new threads which might need to run some form of set up code to properly track metrics for code executed on that thread.

3.2.1 Storage of Dependent Metrics

For storing the dependent metrics during processing, there were a couple of data-structures considered, most notably a fixed-size array and a dynamically-sized array.

The main limitation of the fixed-size array is obviously that the number of metrics that can be tracked at any given time is limited by a compile-time constant. This means that end-users of the tool will not be able to track more metrics than what was decided upon at compile-time. However, considering that the hardware counters that are intended to be tracked in this work are also limited to only a few active ones, it is unlikely that having a fixed upper bound for the number of metrics that could be tracked is impacting the usefulness of the tool. Another potential issue would be that by setting a fixed number of metrics that are supported, it will likely introduce some fixed overhead regardless of how the tool is used. This aspect will be covered in more detail later on in the evaluation in Chapter 4.

In comparison to this, a dynamically-sized array would remove the limitation for the number of metrics being tracked, and it might help to reduce the fixed overhead as well as it could be empty. However, it would likely complicate the implementation as a variety of functions would need to be adapted to handle a new data structure. Having to deal with manual memory management would also increase the burden on maintainers as it represents yet another moving piece that needs to be considered during changes and the dynamic memory allocation could also increase the overhead. Lastly, even though it might reduce the fixed overhead for cases where no metrics are being tracked, it could also increase the overhead whenever metrics are being tracked as the handling of a dynamically-sized array is a bit more involved compared to a fixed-size array.

Given these trade-offs, the decision was made to use a fixed-size array for now. In case the need for more flexibility arises in the future, this part could likely be changed, but it's unclear how much work this would require and what impact it might have on the overhead of the tool.

3.2.2 Functions of the interface

To fulfil the requirements outlined above, the following interface was chosen:

```
class DependableMetric {
public:
  virtual Array<long_long, DEP_METRICS> getStartValue() {
    ...
  }
  virtual Array<long_long, DEP_METRICS> getStopValue() {
    ...
  }

  virtual void registerThread() {
    ...
  }

  virtual char* getMetricName(int idx) {
    ...
  }
};
```

Figure 3.1: Interface for metrics being tracked

All metrics implementations will then extend the *DependableMetric* class and overwrite the virtual functions with the correct implementation for their use-case. Any metric implementation should also provide some kind of set up function or constructor, which is not part of the interface for reasons which will be outlined below.

The *getStartValue* and *getStopValue* methods are the fundamental building blocks of the interface. They are used to retrieve the values of the metrics before and after the application performs some computation. It was chosen to have two distinct functions for getting the starting and stopping values to allow for more flexibility in the metrics being used, as some metrics that are being tracked may need to retrieve values differently in these two cases. In general the values returned from these functions can be arbitrary, however in most cases the value returned by the *getStopValue* function needs to be greater than or equal to the value returned from *getStartValue*, as the difference between them would otherwise be negative, meaning the metric got smaller, which would likely result in confusing or wrong results as we add up all the differences to get the final value.

The *registerThread* function was originally not part of the design. However, it was discovered while implementing the PAPI metrics class, that one needs to set up PAPI for each thread individually, which would not be possible without this function. This function will be called whenever OpenMP first starts a new thread and is responsible for setting up the new thread such that application code running on it will also be properly tracked. Each metric implementation is responsible for

managing their own thread specific details as there is simply too much uncertainty about what exactly different metrics implementations would need to do.

The *getMetricName* function is not strictly necessary for the actual tracking of the critical path or the related dependent metrics. However, it is used to provide a better output at the end of the execution, by providing the rest of the application a way to retrieve the names of the metrics being tracked.

Apart from the details outlined above, there were also a couple of things and features that were considered for the interface as well, but were ultimately not included. Some of these features will be outlined in the next part.

One of the first things to notice is the absence of a general set up or configuration function. The reason behind this is mainly the fact that this would only be needed once to configure the metrics implementation at the start of the program. And since all metrics implementations would need to be checked during the set up anyway, we can call their specific set up functions without any issues. Another reason for excluding this functionality from the interface is the observation that it is currently not clear what such a unified set up function would look like. For example, there is an *EmptyMetrics* implementation which just does nothing and this obviously needs no actual set up and on the other hand, there is the *PAPIMetric* implementation which requires a list of PAPI counters that should be tracked.

Another part that was considered was an analogous function to the internal *maxUpdate* function. This function would have been responsible for updating the local values after synchronizing with a different thread/process. The default implementation for this would simply copy over the values from the chosen critical path and nothing else. There may be cases in which this behaviour should be changed, depending on the metrics being tracked, but as there is no such case implemented or planned right now it was not included in the first iteration of the design.

Related to the previously mentioned absence of a *maxUpdate* function, it was also considered to have a specific function, which would calculate the "usage" of the metrics in a block of computation. Currently, the usage is always calculated as a simple difference between the start and end values of the computation. This approach works in most cases, like for monotonically increasing counters, which is what most hardware counters are. However, if a user were to track something like a pre-calculated rate, this approach would no longer make logical sense, which was the main motivation behind even considering such a thing. In the end this was not implemented as it was not deemed to be critical for this work and most cases that would have a problem with the current approach could be restructured to work with it.

The last detail that was considered is moving the calculation and presentation of the results at the end of the execution into the interface. This would allow metrics

implementations to customize how the metrics collected in each thread and process should be aggregated and presented at the end. The main motivation is that in the future there might be a variety of metrics being supported, and it will likely not make sense to represent their values and results in the same way. For example, if support for tracking the usage in specific sections of the application is added, the current display at the end would likely not be desired and a different representation preferred. In the end, there was no good approach to model the necessary features this would have to support without requiring each implementation to handle all the complexity associated with the final calculation and would likely result in a lot of duplicate code. However, this would be a likely candidate for more experimentation once a metric implementation exists, which could make good use of such a feature.

3.2.3 Interface implementation for PAPI

This section will cover more details about the PAPI implementation of this interface.

To set up the PAPI metrics, the constructor receives a string of PAPI counters which should be tracked. It will then initialize the PAPI library for the current process and parse the provided counter names into their associated codes used by PAPI internally. The names and corresponding codes are also stored in the class, as they are needed to register any new threads with the same values and to provide a readable output at the end. Once this is done, it will actually call the *registerThread* function to set up the current thread, as otherwise the function would not be called for this thread.

The *registerThread* function is relatively straightforward and just checks if the thread is already initialized or not. If it is not, it will create a new PAPI *eventset*, which is how PAPI combines multiple counters into a single interface to start, stop and read from them. Afterwards all the previously parsed counters are added to the thread-local eventset and finally at the end the eventset is started to begin tracking the metrics.

The *getStartValue* and *getStopValue* functions are in this case identical. All they need to do is verify that PAPI is running on the thread, and then perform a simple read from the thread-local eventset. Importantly, this read does not stop or reset the counters, but rather takes a snapshot of their current values and leaves them running.

3.3 Limitations

Here, some of the currently known limitations of the approach outlined above are discussed.

The tool is not capable of knowing which parts of the application are on the critical path. This is because a complete critical path analysis can only be done in a backwards pass and as the OTF-CPT is an online tool by design, this is not possible in the general case. This limitation is reinforced by the fact that the tool is currently not able to track from where in the application code it was called. Theoretically, this could be done by walking the stack to find the stackframes of the caller, however, such an approach would likely incur a lot of overhead and is unlikely to be a suitable approach in an online tool. Furthermore, the tool currently allocates a fixed amount of memory to track the critical path, but it would need to track an arbitrary number of callers, which would not be possible. However, there were some ideas proposed, which would address this limitation to some extent. Specifically, one approach to address this could be to allow application developers to insert specific instrumentation into their application. The instrumentation would then call into the OTF-CPT to mark the start and end of a region of interest, which could allow the tool to determine if these marked regions are on the critical path and potentially how much time is spent in these regions.

Building on the limitation discussed above, the tool is currently also not able to inform the end user about how the different parts of their application contribute to the result they are seeing. Importantly, there is no breakdown for each procedure, like there is in other performance tools. Practically this means that even though the end-user might know that their IPC is rather low on the critical path, they would have no empirical way to determine which part of their application should be optimized, if they only rely on the OTF-CPT.

As previously mentioned, the tool currently has a hard limit for the number of dependent metrics that can be tracked, which needs to be set at compile-time. For the purposes of this work, this does not really represent a problem, as the PAPI counters that can be tracked at any given time are also limited by the hardware itself, and the number of hardware counters that one is usually interested in is also relatively few. However, when considering the fact that we want to support different kind of metrics, this could be an important limitation and drawback of the approach. One example metric that would be limited in its applicability, is the metric discussed above, which aims to keep track of which parts of the code have been called. As each code location would likely correspond to one metric and such, having support for more metrics would directly increase the usefulness of that kind of metric.

Lastly, the tool is also limited to only use one metrics implementation at a time. This is directly caused by only having a single reference to the instance, which is tracking the dependent metrics. Although this specific part could be changed relatively easily to a list of metric implementation that are running at the same time. The complication would likely be in how the available metric "slots" would be distributed and the fact that one would quickly run out of space to store the metrics. The most likely approach would be to have a separate metrics implementation, which

then internally stores two or more other implementations and takes care of handling the metrics. This would likely be the goto approach anyway, as one would likely have a specific use-case in mind, which should help in defining how all of that should be handled.

4 Evaluation and Results

This evaluation mostly focuses on the overhead introduced by the addition of support for dependent metrics, as well as comparing the results obtained from this approach to pre-existing results from other work. The goal is to provide an overview of the overhead introduced and what a user should expect when using the tool. It will also be investigated, if the values obtained from the new approach can replicate the results from other tools and what one might be able to tell from the new dependent metrics.

4.1 Setup

The benchmarks were run on CLAIX-23 [2], where every node consists of two Intel Xeon 8468 Sapphire with 48 cores each and at least 256 gigabytes of RAM. All benchmarks, in which two absolute values are compared, were run on the same node in the cluster directly one after another to ensure that any variance between different nodes would not influence the values differently.

This evaluation will mainly compare the impact of the tool on two applications, a proof of concept juKKR (poc-jukkr) and the corresponding miniapp version of it. The applications implement the Korringa-Kohn-Rostoker (KKR) Greens function method to perform density functional theory calculations. For this purpose, the applications utilize OpenMP to parallelize their computation.

4.2 Overhead

This section will assess the overhead introduced by the tool, as well as having a closer look at how the overhead scales depending on the number of metrics being tracked and supported, as well as how the overhead scales as the number of threads increases. These measurements were also used to set the default values for the tool going forward.

For the following plots, the labels always refer to the following definitions, unless otherwise mentioned. The *Baseline* represents the runtime of the application without any tool used. The *OG-Tool* represents the runtime of the application with the OTF-CPT tool used, before it was modified to support dependent metrics. The *With-Tool* represents the runtime of the application with the OTF-CPT tool used, with support for dependent metrics.

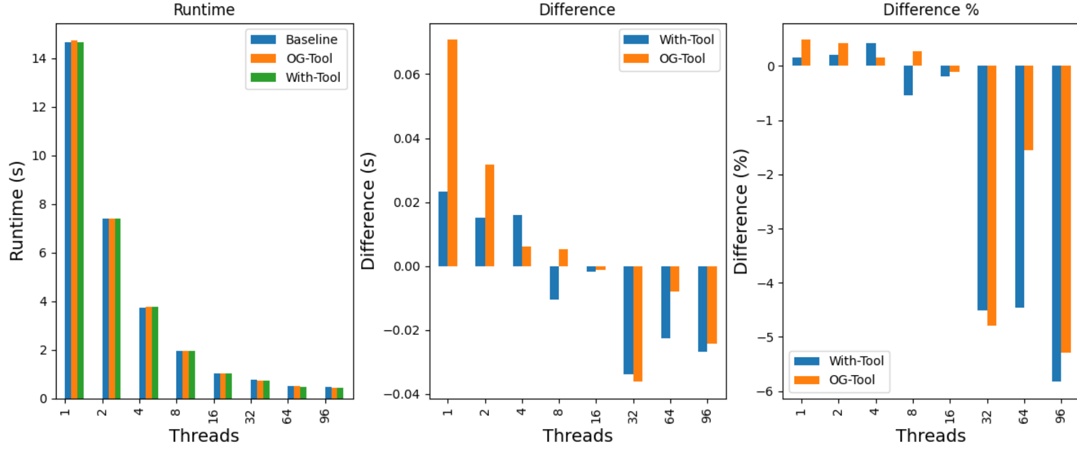

Figure 4.1: Overhead depending on the number of threads used (jukkr-miniapp openmp)

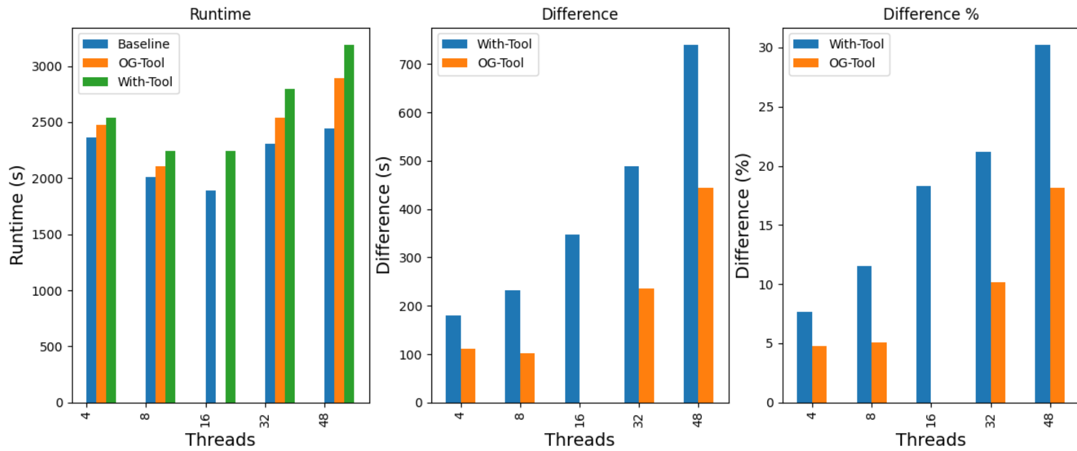

Figure 4.2: Overhead depending on the number of threads used (poc-jukkr openmp)

Figures 4.1 and 4.2 show the runtime of the jukkr-miniapp and poc-jukkr applications depending on the number of threads used and running with no tool, the OTF-CPT without dependent metrics support and the new version while tracking PAPI metrics. For this, the tool was compiled to support 8 dependent metrics and was configured at runtime to track 6 PAPI counters. The runtime was measured in the application itself. Of note is the missing measurements for the poc-jukkr application with 16 threads in Figure 4.2, which is caused by the measurements failing repeatedly. The plots labelled with *Difference* show the absolute and relative difference in runtime of the *OG-Tool* and *With-Tool* compared to the baseline.

The miniapp (Figure 4.1) shows no significant overhead for low thread counts, regardless of the tool version used. However, for larger thread counts, the runtime is getting too short to obtain reliable measurements, which causes noise from outside sources to become more significant as it shows the runs with a tool to be faster.

This is very likely not true and just a consequence of the aforementioned noise, and should more likely be interpreted as being effectively the same.

Looking at the overhead for the entire application (Figure 4.2), it clearly shows the overhead increasing with more threads, for both versions of the tool. The difference in overhead, between the two versions of the OTF-CPT, seems to stay at a constant factor of around 2x. However, it is important to note that the application was not initially designed with higher than around 16 threads per process in mind. This clearly shows in the overall runtime, which steadily decreases while going from 4 to 16 threads, but then quickly grows longer again when using more than 16 threads. One potential explanation for this may be that the work to be done is getting too small when distributed across many threads and therefore the actual computation is getting short enough that the fixed or nearly fixed overhead of calling into OpenMP functions and the OTF-CPT takes up a significant portion of the time. Nonetheless, the overhead of the tool is not insignificant even when mainly looking at the lower thread counts.

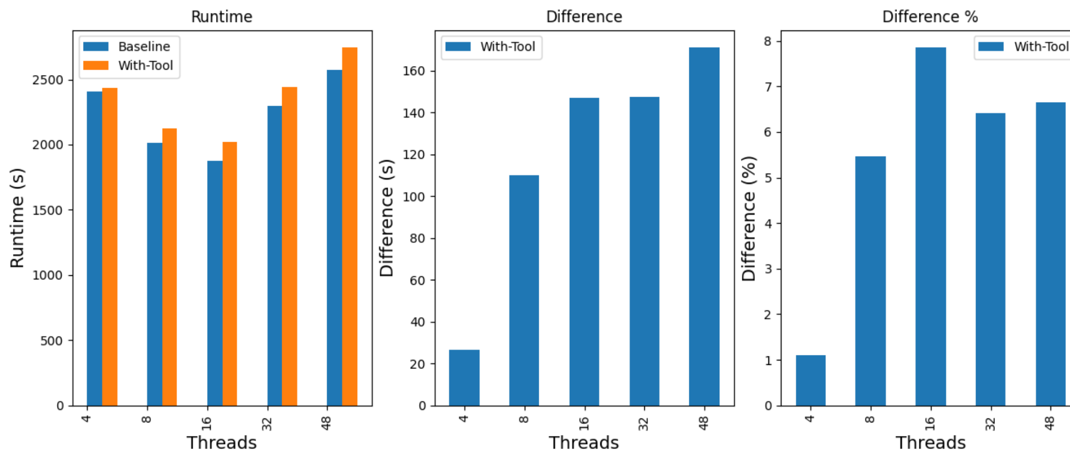

Figure 4.3: Overhead for ScoreP Run (poc-jukkr)

Figure 4.3 shows the runtime of the poc-jukkr application with and without using Score-P[8]. For this case, Score-P was configured to track the same 6 PAPI counter as the OTF-CPT in the measurements above, and the runtime is once again measured by the application itself.

In this case, Score-P appears to have a significantly lower overhead compared to the OTF-CPT (Figure 4.2) and also does not exhibit the same increase in overhead. However, as Score-P is a post-processing tool, it does not perform the critical path analysis and the collected data still needs to be processed using other tools, like Scalasca[12]. The time and effort that would be associated with the extra steps is not reflected in these results, as it would be hard to define what exactly one would measure for this and one could use different tools to perform the processing. Additionally, Score-P also imposes a potentially significant memory and storage overhead, as it needs to store all the collected data first in memory and then write it out to disk. While performing the benchmarks, Score-P estimated the memory

requirements to be between 2GB and 30GB and the size for event trace was estimated to be between 2GB and 29GB.

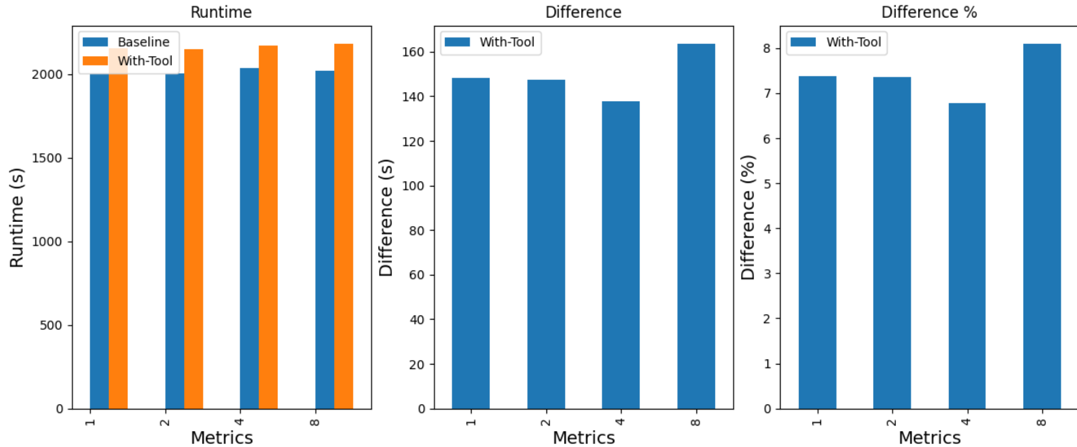

Figure 4.4: Overhead depending on the number of supported metrics (no metrics used, 8 threads, poc-jukkr)

Figure 4.4 shows the runtime overhead for a varying number of supported dependent metrics compared to running the poc-jukkr application without any tool. The measurements were performed with 8 threads and no dependent metrics being tracked to isolate the overhead introduced by just having the support for them in the tool.

The observed overhead is relatively constant with only minor variance, which can likely be attributed to noise. This is likely a result of the tool only operating on the number of tracked metrics for most operations. Specifically, the *maxUpdate* function, which is responsible for copying the values, only operates on the number of tracked metrics. However, as this is only measuring the overhead for OpenMP applications, the overhead when using MPI is currently not known and is likely higher as the code responsible for sending the data is always sending all the possible metrics.

Based on this result, it was decided to support 8 dependent metrics by default, as the overhead shows no significant increase compared to even just 1 supported metric. At the same time 8 dependent metrics seems like enough for most basic metrics, especially for PAPI counters as tracked in this work, as they are inherently limited to around 8 by the hardware.

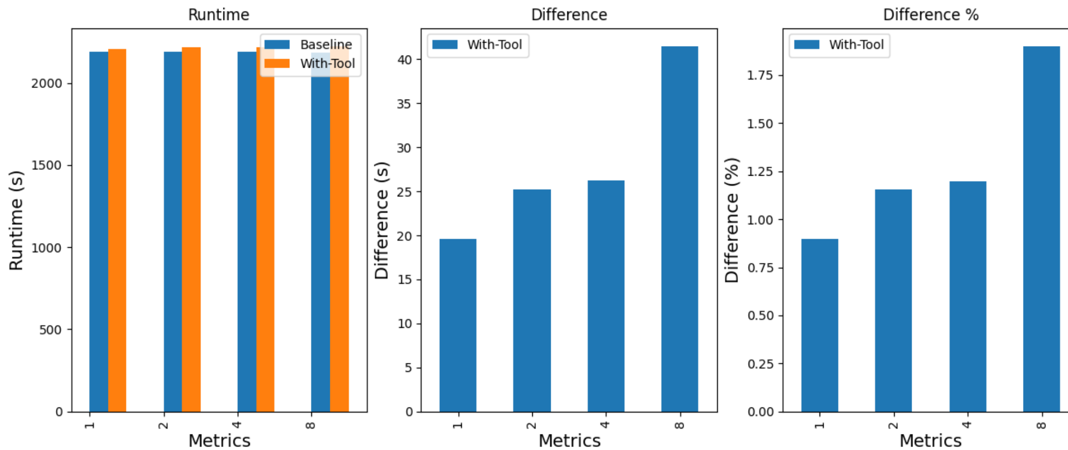

Figure 4.5: Overhead depending on the number of used metrics (but same number of supported metrics, 8 threads, poc-jukkr)

Figure 4.5 shows the runtime overhead depending on the number of tracked PAPI counters, compared to running the poc-jukkr application with the OTF-CPT supporting 8 dependent metrics but tracking none. This mostly isolates the overhead of the implementation to track the PAPI counters, as well as the overhead of PAPI itself.

As can be seen in the plots, the additional overhead for using the PAPI metrics is relatively low, at around 0.8% to 1.8%. There is a trend for the overhead to increase with the number of metrics being tracked, which is the expected behaviour as there is more work to do. The increase is likely related to how the values are updated in the *maxUpdate* function, because it only copies the number of metrics that are being tracked. This means that for this specific part of the code, it does 8 times more work when tracking 8 metrics compared to when it is only tracking 1.

4.3 Comparison to existing results

This section will compare results obtained from using the OTF-CPT with results obtained from Score-P. It will first compare the measurements for the entire application, which are obtained by summing up the values of each thread. And then compare them with the measurements along the critical path, calculated based on the useful computation.

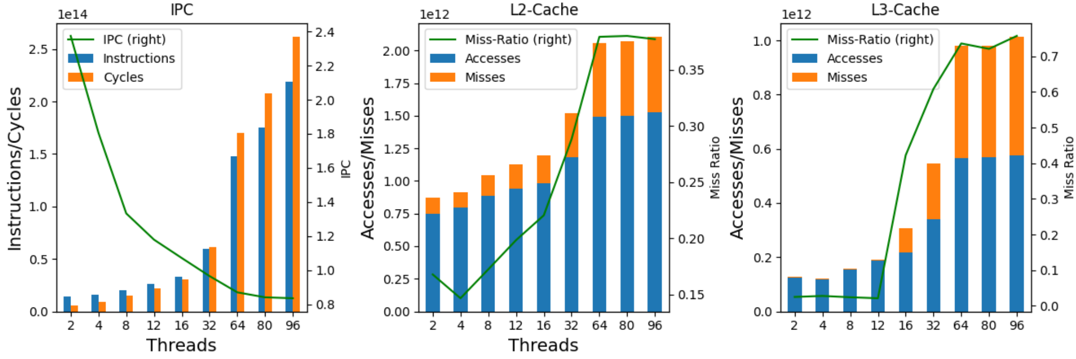

Figure 4.6: Score-P measurements (poc-jukkr)

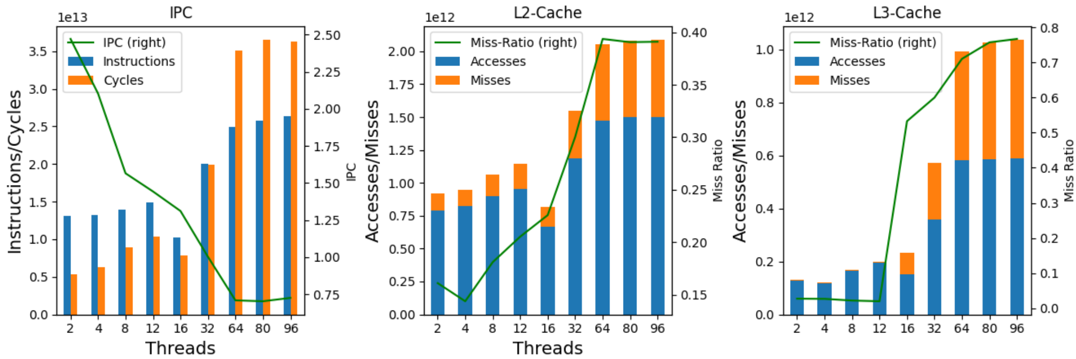

Figure 4.7: OTF-CPT measurements for entire application (poc-jukkr)

Figure 4.6 and 4.7 show the metrics measured over the entire poc-jukkr application, which means the values are summed up over all the threads. This mostly represents the data that one would obtain from a standard performance analysis.

The plots for the L2-Cache and L3-Cache measurements in figure 4.6 and 4.7 correlate very well and only have minor differences in the values. However, the absolute values for the instruction and cycle counts differ significantly between the two figures. It is unclear what exactly causes this to be the case, especially seeing as the calculated IPC behaves mostly the same between the two plots. However, it might be related to how the values are calculated and categorized by the tools and in this case, Score-P seems to include a lot more in their calculations.

As previously mentioned, the application was not designed for higher thread counts and this can also be seen in the plots, as most measures get significantly worse for 32 threads and above.

The measured IPC follows a clear downwards trend as the number of threads increases and levels out at around 0.8 for 64 threads. While the IPC decreases, the L2-Cache miss ratio tends to increase quickly and levels out at 0.35 for 64 threads as well. The L3-Cache miss ratio stays relatively constant at around 0.05 for up to 12 threads, after which it rapidly increases up to about 0.75 for 64 threads.

From these observations, it seems like the main problem is the cache usage and access pattern, as both cache miss ratios increase and get worse while the IPC decreases. First this is mostly happening in the L2-Cache while the miss ratio for the L3-Cache stays constant up until 16 threads, after which it quickly increases. At that point the application can likely no longer make efficient use of the L3-Cache and all cache misses have to go to RAM, which is significantly slower, which would explain the very sudden drop in IPC.

Based on the strong similarities between the values obtained from the two tools, the values produced by OTF-CPT can be assumed to be correct for this use-case.

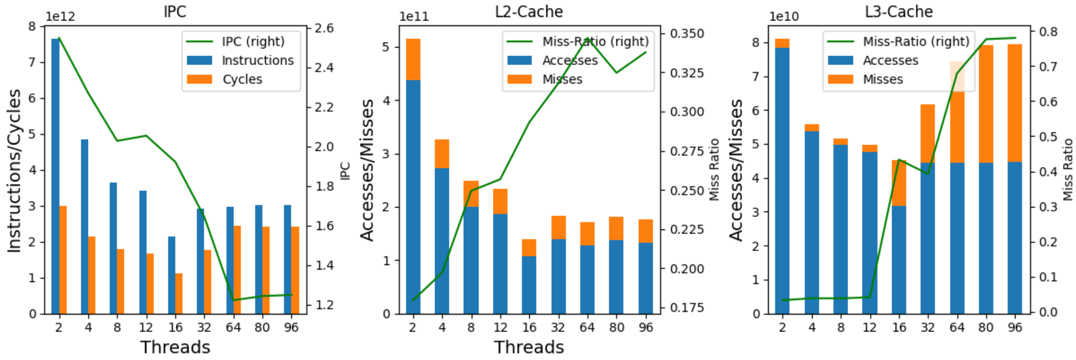

Figure 4.8: OTF-CPT measurements along the critical path (poc-jukkr)

Figure 4.8 shows the same metrics as can be seen in figures 4.7 and 4.6, but only tracked along the critical path of the application, based on the useful computation.

One of the main differences between Figure 4.8 and Figure 4.7 is that the values for the critical path are effectively only for one thread and not summed up over all the threads of the application. This is because the critical path is just a sequence of computations, and in that sense the values can be viewed as if they are from just a single thread.

The initial observation is that the number of instructions, cycles, L2-Cache accesses and L3-Cache accesses decrease as the number of threads increase, up to about 16 threads. At that point the instructions, L2-Cache accesses and L3-Cache accesses reach a steady level and stay around that value even as the number of threads increases. However, after 16 threads the cycles, L2-Cache misses and L3-Cache misses increase with the number of threads.

In terms of the IPC, L2-Cache miss ratio and L3-Cache miss ratio, the plots from figure 4.8 are very similar to the plots in figure 4.7. And in general, one could come to the same conclusion based on the values from the critical path as the conclusion for the entire application.

However, the critical path values can also provide some more potential reasons. For instance, the number of instructions executed on the critical path decreases with higher thread counts is in line with the expectations, as the same work is distributed across more threads and therefore each individual thread has to do less. But this trend does not continue forever, and the levelling out suggests that either the work

can no longer be split up into smaller units to work on or that the application for some reason is not distributing the total amount of work evenly.

This shows that the values obtained for the critical path can provide new insights and aid in understanding the performance characteristics of the application.

5 Conclusion and Future Work

The general goal to be able to track arbitrary metrics on the critical path has been reached. Specifically, the OTF-CPT now supports tracking arbitrary metrics and providing their respective results for the entire application or just the critical path. Even though this thesis implements PAPI counters as the only type of available metric so far, it provides the necessary structure to implement more metrics in the future.

The evaluation shows that the metrics measured by the OTF-CPT can be useful for investigating an applications' performance. The measurements for an entire application seem to be correct and can replicate the values one would obtain from using a tool combination like Score-P and Scalasca, however the absolute values might differ between tools. Importantly, the metrics for the critical path can provide insights that were not obvious from just looking at the values for the entire application. As was seen in the evaluation, in which the values for the critical path have shown that the work per thread reaches a plateau, after which adding more threads does not improve the runtime.

The observed runtime overhead of the tool can vary widely depending on the characteristics of the underlying application. For applications that rarely call into OpenMP or MPI, the overhead should be fairly minimal, as the tool does not get called as often. However, in cases where there are very frequent calls to OpenMP or MPI the overhead can become more significant, as observed with the poc-jukkr application for higher thread counts.

In regard to potential future work, the first starting point could be to extend the OTF-CPT with support for more metrics to be tracked. One such metric could be the idea outlined in chapter 3.3 regarding special instrumentation to test if a specific region of code is on the critical path and how much computation time that region contributed to said critical path. This kind of metric would involve more work from the application developers, compared to using the PAPI metrics, but could provide very beneficial insights into the characteristics of the application and help determine which parts of the application would benefit most from being improved.

One could also investigate the feasibility of using dynamically sized storage for the dependent metrics, as outlined in section 3.2.1. Such an extension could be very useful in removing one of the limitations, however there would need to be paid close attention to the overhead introduced by such a change, as it is already higher than initially expected.

One last consideration could be to investigate ways to reduce the overhead of the OTF-CPT, as reducing the baseline overhead could then be used to increase the

number of metrics being supported, while still being practical to use.

Bibliography

[1] David Böhme et al. “Scalable Critical-Path Based Performance Analysis”. In: *Proc. of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, China*. IEEE, May 2012, pp. 1330–1340. DOI: 10.1109/IPDPS.2012.120.

[2] *CLAIX-2023*. 2023. URL: https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/.

[3] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[4] Alexandre E. Eichenberger et al. “OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis”. In: *OpenMP in the Era of Low Power Devices and Accelerators*. Ed. by Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185. ISBN: 978-3-642-40698-0.

[5] *Hardware events for Intel Sapphire Rapids*. 2025. URL: https://perfmon-events.intel.com/spxeon.html.

[6] Joachim Jenke et al. “A Shim Layer for Transparently Adding Meta Data to MPI Handles”. In: *Proceedings of the 30th European MPI Users’ Group Meeting*. EuroMPI ’23. Bristol, United Kingdom: Association for Computing Machinery, 2023. ISBN: 9798400709135. DOI: 10.1145/3615318.3615324. URL: https://doi.org/10.1145/3615318.3615324.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[8] Dieter an Mey et al. “Score-P: A Unified Performance Measurement System for Petascale Applications”. In: *Competence in High Performance Computing 2010*. Ed. by Christian Bischof et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–97. ISBN: 978-3-642-24025-6.

[9] Philip J Mucci et al. “PAPI: A portable interface to hardware performance counters”. In: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710. Citeseer. 1999.

[10] *OpenMP 6.0*. 2024. URL: https://www.openmp.org/.

[11] Joachim Protze et al. “On-the-Fly Calculation of Model Factors for Multi-paradigm Applications”. In: *Euro-Par 2022: Parallel Processing*. Ed. by Cano et al. Cham: Springer International Publishing, 2022, pp. 69–84. ISBN: 978-3-031-12597-3.

[12] *Scalasca*. 2025. URL: https://www.scalasca.org/.

[13] Frank Wrinkler. “Redesigning PAPI’s high-level API”. In: *University of Tennessee, Tech. Rep. ICL-UT-20–03* (2020).

[14] C-Q Yang and Barton P Miller. “Critical path analysis for the execution of parallel and distributed programs”. In: *The 8th International Conference on Distributed*. IEEE Computer Society. 1988, pp. 366–367.